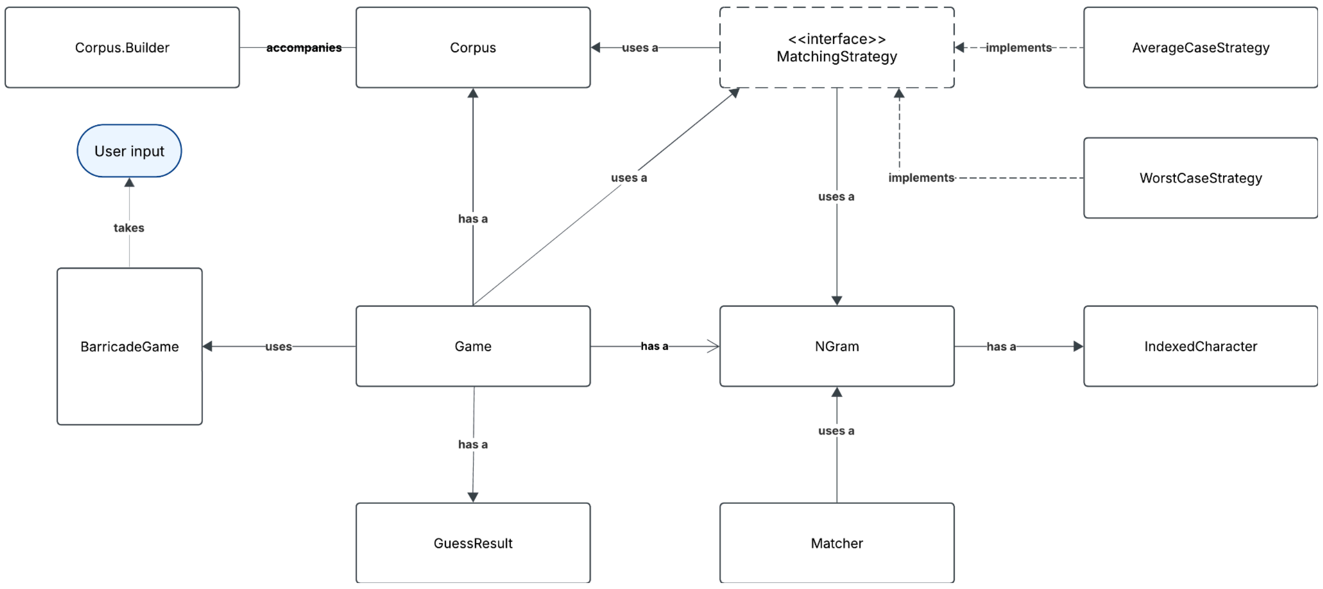
Thao Nguyen ttn60

CSDS 293 Homework 11

Matchle Design Document

General Outline



Abstractions:

1	Game	Central class that interacts with all other components. It has methods for making guesses, checking if the game is over, and tracking history.
2	NGram	An immutable sequence of characters with indexed positions.

		Models a 'word' of fixed length.
3	IndexedCharacter	A character paired with its position (index).
4	Corpus	A dictionary of ngrams (both keys and guesses has to be in this dictionary)
5	Matcher	Encapsulates a predicate and provides a unified interface to evaluate NGram matches
6	MatchingStrategy	Different guessing strategies that can be employed by the player to get closer to the key, based on the previous guess feedback.
7	WorstCaseStrategy	Implements MatchingStrategy, come up with the guess that results in the smallest corpus size when the key is the worst possible one.
8	BestCaseStrategy	A strategy where the player guesses the n-gram that results in the smallest average corpus size.

9	GuessResult	A guess made by the player and its corresponding result for the NGram as a whole (matched or not matched to the key) and for each character (exists in the key and in the correct spot, exists in the key and in the incorrect spot, or does not exist in the key).
10	BarricadeGame	Validate player input before passing it to the game.

Class Design

1. Game

Attributes:

- **key** (NGram): The hidden word that the player is trying to guess.
- **history** (List<GuessResult>): A list tracking each guess made by the player and its corresponding result.
- **maxGuesses** (int): The maximum number of guesses allowed before the game is over.
- **corpus** (Corpus): A dictionary of valid guesses for the game.

Routines:

1. **makeGuess(String guess)**
    - **Input**: String guess

        - A player’s guess.

    - **Return Type**: void

    - **Preconditions**: guess must be a valid word according to the dictionary.

    - **Postconditions**: The guess is recorded in the history, and the game state is updated. If the guess is correct, the game may end.

    - **Algorithm**:

        - Check if the guess is valid.

        - Use the Matcher to compare the guess against the key.

        - Store the result in history.

        - Check if the game is over (win or max guesses reached).

    - **Error Handling**:

        - Throw IllegalArgumentException if the guess is not valid or if the game has already ended.

        - Validate that the input guess exists in the dictionary.

2. **makeGuess(MatchingStrategy strategy)**
    - Generate a guess with one of the provided or customly defined strategies and the history of guesses, and make that generated guess.

3. **isOver()**
    - **Input**: None

    - **Return Type**: boolean Returns true if the game is over (win or loss).

    - **Preconditions**: None
    - **Postconditions**: None
    - **Algorithm**:
        - Check if the game has reached the maximum number of guesses or if the player has won.

4. **isWin()**
    - **Input**: None
    - **Return Type**: boolean
        - Returns true if the player has won (made the correct guess).
    - **Preconditions**: None
    - **Postconditions**: None
    - **Algorithm**:
        - Return true if the last guess in history is a perfect match.

5. **getHistory()**
    - **Input**: None
    - **Return Type**: List<GuessResult>
        - Returns the history of guesses.
    - **Preconditions**: None
    - **Postconditions**: None
    - **Algorithm**:
        - Return the list of GuessResult objects.

Error Handling:

- **Invalid Guess**: Ensure guesses are valid words from the dictionary, using the BarricateGame to validate input first.
- **Game Over**: Prevent further guesses once the game is over.

Unit Tests:

1. **testMakeGuess()**:
   - Test valid and invalid guesses.
   - Test after reaching max guesses.
2. **testIsOver()**:
   - Test when the game is over (win or loss).
3. **testIsWin()**:
   - Test win scenario (correct guess).
4. **testGetHistory()**:
   - Test history tracking after several guesses.

2. NGram

Attributes:

- **ngram** (List<IndexedCharacter>): A list of indexed characters.

Routines:

1. **size()**
   - **Input**: None
   - **Return Type**: int The length of the NGram.

   - **Preconditions**: None
   - **Postconditions**: None
   - **Algorithm**:
     - Return the size of the value list.

2. **charAt(int index)**
   - **Input**: int index
     - The index to retrieve.
   - **Return Type**: char
     - The character at the given index.
   - **Preconditions**: index must be within bounds.
   - **Postconditions**: None
   - **Algorithm**:
     - Return the character at the specified index.
3. **stream()**
   - **Return Type**: boolean
     - Returns a stream of IndexedCharacter in the ngram.
4. **matches(IndexedCharacter ic)**
   - **Input**: IndexedCharacter ic
     - The character and index to compare.
   - **Return Type**: boolean
     - Returns true if the NGram matches the indexed character at the same index.
   - **Preconditions**: None
   - **Postconditions**: None

    - **Algorithm**: Compare the character at the specified index with the character in ic.

5. **contains(char c)**

    - **Input**: char c
        - The character to check.

    - **Return Type**: boolean
        - Returns true if c exists in the NGram.

    - **Preconditions**: None

    - **Postconditions**: None

    - **Algorithm**:

        - Check if the character c exists in value.

6. **containsElseWhere(IndexedCharacter ic)**

    - **Input**: IndexedCharacter ic
        - The character and index to check.

    - **Return Type**: boolean
        - Returns true if the NGram has the indexed character at a different index.

    - **Preconditions**: None

    - **Postconditions**: None

    - **Algorithm**:

        - Compare the character at the specified index with the character in ic.

Error Handling:

- **Index Out of Bounds**: Ensure indices are within bounds when doing operations on the characters.

Unit Tests:

1. **testLength()**:
    - Test the length calculation.
2. **testCharAt()**:
    - Test character retrieval by index.
3. **testContains()**:
    - Test character existence.
4. **testMatches()**:
    - Test matching between IndexedCharacter and NGram.

3. IndexedCharacter

Attributes:

- **index** (int): The index of the character.
- **character** (Character): The character.

Routines:

1. **int index()**
    - Getter for index
2. **Character character()**
    - Getter for character

4. Corpus

Attributes:

- **corpus** (List<NGram>): A list of valid NGram words in the dictionary. All NGram objects in this list must have the same length.

- **wordSize** (int): The fixed length of the NGram words in the dictionary. This length must be consistent across all NGram objects in the dictionary.

Routines:

1. **contains(NGram word)**

   - **Input**: NGram word The NGram object to check for existence in the dictionary.

   - **Return Type**: boolean Returns true if the word exists in the dictionary, false otherwise.

   - **Preconditions**: word must not be null.

   - **Postconditions**: None

   - **Algorithm**:

     - Check if the word exists in the words list. Return true if found, otherwise return false.

3. **wordSize()**

   - **Input**: None

   - **Return Type**: int
     - Returns the length of the words in the dictionary.

   - **Preconditions**: None

   - **Postconditions**: None

   - **Algorithm**:

        - Return the wordLength attribute.

4. **validateWord(NGram word)**

    - **Input**: NGram word
        - The NGram word to validate.

    - **Return Type**: boolean
        - Returns true if the word is valid (i.e., exists in the dictionary and matches the required length), false otherwise.

    - **Preconditions**: word must not be null.

    - **Postconditions**: None

    - **Algorithm**:

        - Check if the word exists in the dictionary (contains(word)) and if it matches the wordLength.

        - Return true if both conditions are met; otherwise, return false.

5. **score(NGram guess, Function<LongStream, Long> aggregator)**
    - **Inputs**
        - guess: The guessed NGram.
        - aggregator: A function to reduce a stream of scores (e.g., max, sum).
    - **Return Type**
        - long: The aggregated score using the provided aggregation function.
    - **Preconditions**
        - guess and aggregator must not be null.
    - **Postconditions**
        - Returns a long value produced by applying the aggregator on all scores against the guess.
    - **Algorithm Description**
        - For each key in corpus, calculate score(key, guess).
        - Aggregate those values using aggregator.
    - **Error Handling**
        - Throws AssertionError if inputs are null.

        - Aggregator must be safe against empty streams if used externally

Subclass: Builder

- A companion mutable class to build the corpus
- Field
    - ngrams: Set<NGram>
        - the candicate corpus
- Methods:
    1. **add(NGram word)**

        - **Input**: NGram word
            - An NGram object representing a word to be added to the dictionary.

        - **Return Type**: boolean
            - Returns true if the word was successfully added, false otherwise.

        - **Preconditions**: word must not be null and must have the same length as wordLength.

        - **Postconditions**: The word is added to the words list if it passes validation.

        - **Algorithm**:

            - Validate that the word is not null and that its length matches wordLength.

            - If valid, add the word to the words list and return true.

            - If the word doesn't meet the validation criteria, return false.

    2. isConsistent()

        - **Return Type**: boolean
            - Returns true if the ngrams are consistent in length

    3. build()

        - **Return Type**: Corpus
            - Returns a Corpus if the ngrams are consistent in length

Error Handling:

- **Invalid Word**: Ensure that only NGram objects with the correct length are added to the dictionary. If a word has an invalid length, it will not be added.

- **Null Input**: Prevent null values from being added or checked in the dictionary.

Unit Tests:

1. **testAdd()**:
    - Test adding valid and invalid words (NGram) to the dictionary.
    - Ensure that only words of the correct length are added.
2. **testContains()**:
    - Test checking for the existence of a word in the dictionary.
3. **testValidateWord()**:
    - Test validating a word for correctness, including checking if it is in the dictionary and has the correct length.
4. **testGetWordLength()**:
    - Test that the correct length is returned for words in the dictionary.

5. Matcher

Fields:

- private NGram key
- private NGram guess

Routines:

1. **match()**
    - **Return Type**: GuessResult
    - **Preconditions**: Both key and guess must be valid.
    - **Postconditions**: Returns a GuessResult.
    - **Algorithm**:
        - Compare the guess to the key and create a GuessResult based on the match type for each character (correct, present, absent).

Error Handling:

- **Invalid Input**: Ensure the key and guess are not null.

Unit Tests:

1. **testMatch()**:
    - Test with various guesses and key combinations.

6. MatchingStrategy (Interface)

Field:

- GuessResult guessResult

Routines:

1. NGram chooseGuess(Corpus corpus)

7. WorstCaseStrategy

(implements MatchingStrategy)

- Calculate the maximum score among all corpus' n-grams in terms of how much it fits the guess result, using Corpus.score() with the function LongStream.max()
- Pick the NGram with that maximum score to be the guess.

8. AverageCaseStrategy

(implements MatchingStrategy)

- Calculate the average score among all corpus' n-grams in terms of how much it fits the guess result, using Corpus.score() with the function LongStream.sum() and dividing by the number of ngrams.
- Pick the NGram closest to that average score to be the guess.

9. GuessResult

Fields:

- NGram guess
- Map<IndexedCharacter, MatchType> resultMap

MatchType: enum EXACT, PARTIAL, NONE

Routines:

- boolean isMatch() true if all characters are EXACT.

Unit Tests:

- testIsMatch_correct
- testResultMap_accuracy

10. BarricadeGame

Fields:

- Game game: The internal instance of actual game logic

Routines:

1. **makeGuess(String rawInput)**
    - **Input**: The user's guess as a raw string
    - **Return Type:** void
    - **Preconditions:**
        - Game is not over
        - String is non-null and non-empty
    - **Postconditions:**

        - If valid, the guess is passed to the game's makeGuess logic.
        - If invalid, an appropriate exception or user-friendly error is thrown/logged.
2. **validateInput (String guess)**
    - **Input**: String guess
    - **Return Type**: NGram validatedGuess
    - Helper method for makeGuess, returning a cleaned version of the NGram after substituting values.
3. **getHistory(), isOver(), isWin()**
    a. Delegate to Game class through the game instance.

Error Handling

- Throws IllegalArgumentException with specific feedback (e.g., "Word not in dictionary", "Incorrect length").
- Substitute or suggest the user with the closest valid word.
- Logs malformed inputs for debugging.
- Uses defensive checks for nulls and game state.

Stress Test

- Generates large NGram words (lengths ~12) and large Corpus (size ~ 300000).
- Times the matcher's execution and ensures consistent results.
- Verifies that execution completes within a reasonable time frame.